Portfolio 2: Simple Pong

Group 20

Matt McKillip | Ryan Smelser | Phil Goodman

10/26/14

Table of Contents

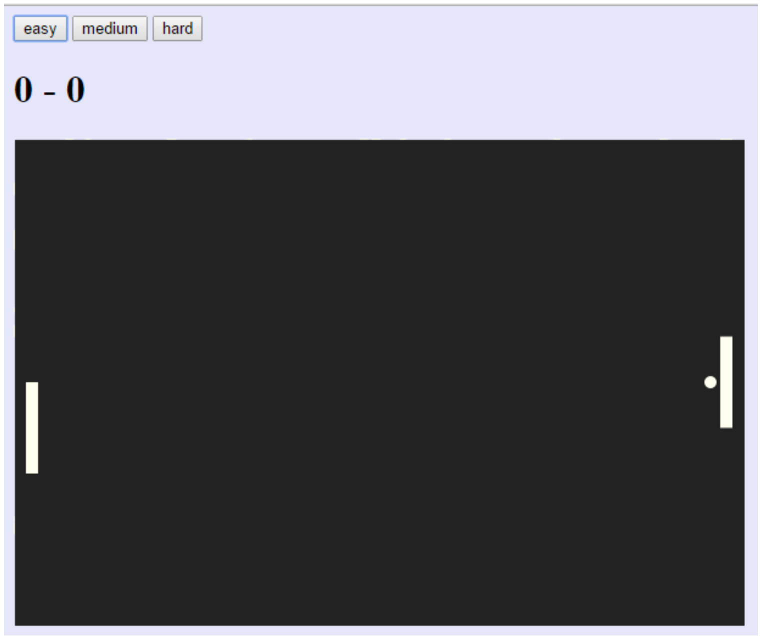
Overview

For portfolio 2 we decided to demonstrate our JavaScript knowledge by implementing JavaScript objects, animation, closure, and php MySQL database creation. To show this, we have created our own version of the famous Pong game. Our implementation of Pong builds off of the topics covered in class and lab. In creating this game we focused on four complex topics:

1. JavaScript Prototypes
2. JavaScript Animation
3. JavaScript Closures
4. MySQL database creation

We focused on these issues upon other simpler issues when creating our game. We then focused on analyzing and evaluating our code to find areas of improvement. In our Bloom’s Taxonomy section we focus on how we created code that was fast, maintainable and scalable.

Screenshot



External References

http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/

http://sporto.github.io/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/

http://javascriptweblog.wordpress.com/2010/06/07/understanding-javascript-prototypes/

http://creativejs.com/resources/requestanimationframe/

http://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

http://www.w3schools.com/php/default.asp

http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/

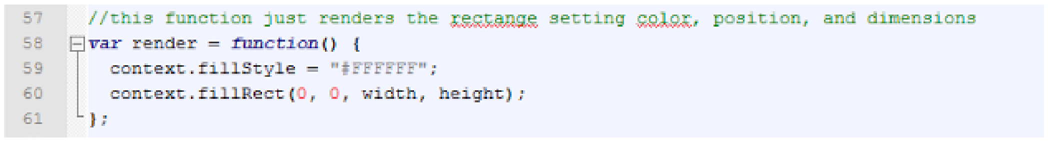
New and Complex Issues

JavaScript Prototypes

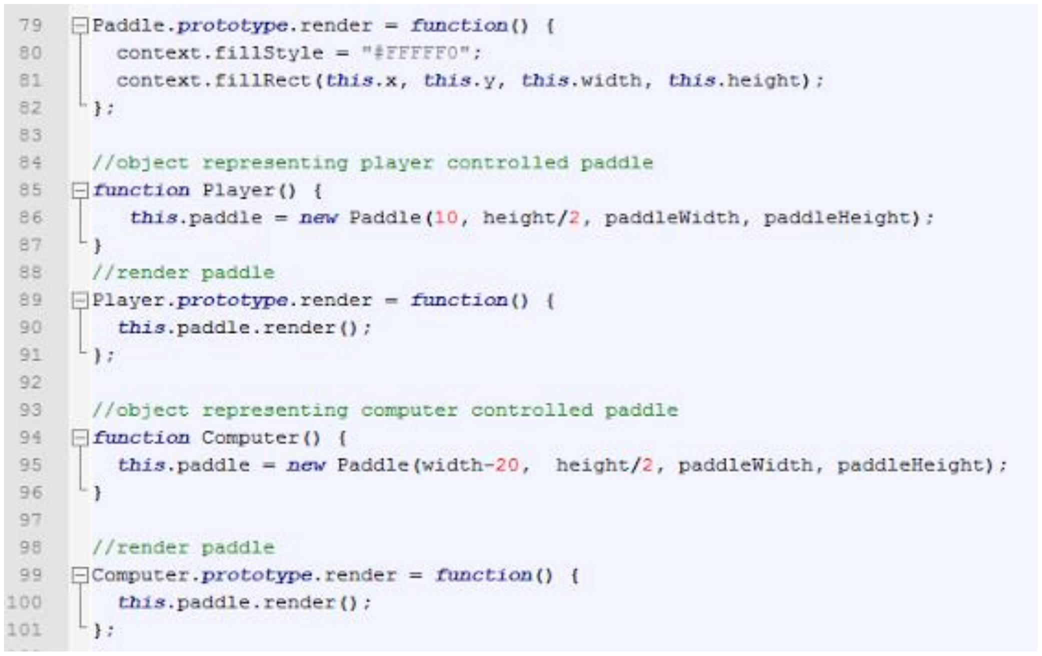
JavaScript is unique to many languages due to its object construction. Since JavaScript makes no distinction between constructors and other functions like many object oriented programs do, every function must have a prototype property.

The reason why is because JavaScript does not have object inheritance for classes like other object oriented programming languages. Prototypes provide a useful work around to make a “parent” object of functions that act as objects, much like a parent class in Java.

The reason we decided to use prototypes in our implementation was our objects had many similar methods. For instance both our Computer and Player objects have a render function. Below is a snippet showing the implementation of adding a prototype render to the Computer object.



```
//this function just renders the rectange setting color, position, and dimensions
var render = function() {
    context.fillStyle = "#FFFFFF";
    context.fillRect(0, 0, width, height);
};
```



```
Paddle.prototype.render = function() {
    context.fillStyle = "#FFFFF0";
    context.fillRect(this.x, this.y, this.width, this.height);
};

//object representing player controlled paddle
function Player() {
     this.paddle = new Paddle(10, height/2, paddleWidth, paddleHeight);
}
//render paddle
Player.prototype.render = function() {
    this.paddle.render();
};

//object representing computer controlled paddle
function Computer() {
    this.paddle = new Paddle(width-20,  height/2, paddleWidth, paddleHeight);
}

//render paddle
Computer.prototype.render = function() {
    this.paddle.render();
};
```

Using prototypes was a new topic for our group. We have had no JavaScript experience outside of the exercises, so we had to do research in the best way to complete this task. We found prototypes to be our solution. We are using prototypes like we would use a parent class. Our code has several examples similar to the one above.

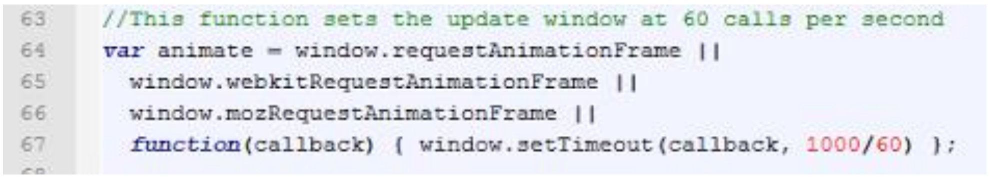
The biggest issue that we had implementing prototypes was distinguishing the difference between using prototypes and *this*. In lab our JavaScript code used *this*, but we decided to do some research to see if there was a more efficient way to assign properties to an object. What we discovered was using *this* results in every instance of the object to have its own independent copy of the method, but using prototype means that every instance of the object has the same copy of the method. While in our simple game, using *this* would have been feasible. Our thought was using prototypes would be more maintainable and scalable.

JavaScript Animations

When creating a game in JavaScript correctly implementing animation is crucial. In Exercise 03 our group implemented a simple JavaScript snake game. In this lab we used a timer function that used setInterval to drive the animation. This is a very quick and simple way to do JavaScript animation, but our research found a better way to implement animation.

What is wrong with this simple animation technique is that setTimeout does not take into account what is happening in the browser. If the canvas is not in focus - different window, tab, or scrolled - setTimeout will keep using up resources when it is not needed. This can slow down the performance of your browser due to unnecessary CPU computations.

To combat this problem we found a solution in Mozilla's requestAnimationFrame function. This function allows the browser to ignore animation requests when the canvas is not in focus. Additionally it optimizes the setTimeout with the canvas refresh, also increasing browser performance. Below is our implementation of requestAnimationFrame:



```
//This function sets the update window at 60 calls per second
var animate = window.requestAnimationFrame ||
  window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame ||
  function(callback) { window.setTimeout(callback, 1000/60) };
```

We chose to include window.webkitRequestAnimationFrame and window.mozRequestAnimationFrame because it is more robust solution supporting different browsers. While requestAnimationFrame should work on most modern browsers we included webkitRequestAnimationFrame which supports Chrome and Safari, and mozRequestAnimationFrame which supports Firefox.

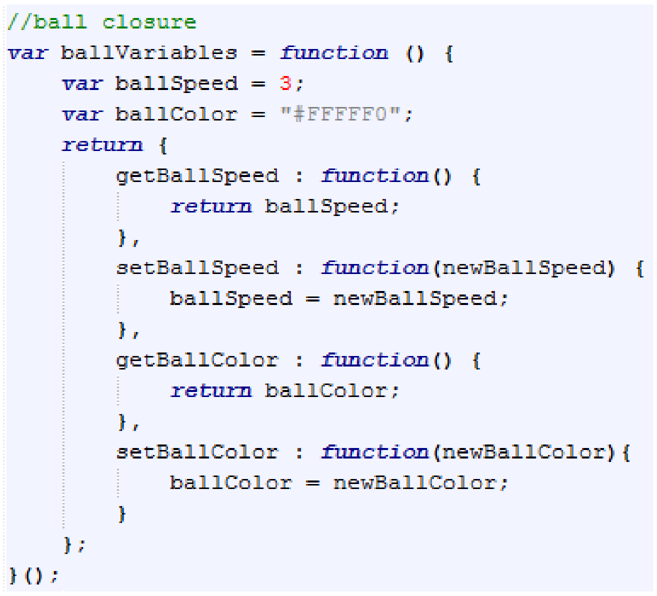
Choosing to find a better implementation of animation may not increase the performance of our simple game, but would be critical in a more advanced JavaScript animation implementation. We chose to use it because it is good practice to consider maintainability and scalability even when creating a simple project.

JavaScript Closures

When writing any type of JavaScript code it is important to always consider if closures are appropriate to hide any functions or variables from additional JavaScript's and keep them to a local scope. We originally explored closures in exercise 4, learned several benefits of using them and what they can and cannot do. We were still confused about the general idea of how to appropriately use closures and in this portfolio project we were able to successfully use them in a more correct fashion.

At a basic level closures are functions that define variables, do work with those variables, return whatever work was done and successfully keep the internal variables and any other functions private. In the calculator exercise we used a closure to hide the memory value. In this portfolio we had a lot of global variables we were using for the canvas, paddles, ball and scores. Leaving access to these variables open to all other methods and JavaScript files would definitely be problematic. Tampering with these variables would likely crash or break the game, or allow players to cheat, and barring any malicious intent there's no reason to muck up the variable pool with these global variables in case the JavaScript was to be used as part of another site.

The full code of closures can easily be seen implemented at the beginning of our JavaScript file, and one sample from our JavaScript should suffice to explain what we did:



```
//ball closure
var ballVariables = function () {
    var ballSpeed = 3;
    var ballColor = "#FFFFF0";
    return {
        getBallSpeed : function() {
            return ballSpeed;
        },
        setBallSpeed : function(newBallSpeed) {
            ballSpeed = newBallSpeed;
        },
        getBallColor : function() {
            return ballColor;
        },
        setBallColor : function(newBallColor){
            ballColor = newBallColor;
        }
    };
}();
```

In our closures we initiated the variables (in this case the ball properties) and included a get() function for each variable when the closure is called, and sometimes a set() function if it was appropriate. The inner variables cannot be simply accessed by a call like ballVariables.ballSpeed; that would return "undefined". To access the variable a ballVariables.getBallSpeed () call must be used, protecting the variables from other methods and JavaScript's. The material we used as a guide can be found via the hyperlink in our references.

MySQL Database Creation

Continuing under the premise that this application would be implemented as an online game, it would be desirable to maintain records containing information about the individual players and games. MySQL databases could serve as a simple way to manage these records. PHP scripts on the server-side would then be used to query the databases and offer valuable information to the user.

Every player would be directed to a sign-up or log-in screen, similar to Exercise 6. Information about the user could be stored in a “users” table, shown below:

UserID	Password	Email

The “UserID” column would serve as the primary key for this table, so that a player would receive an error message if they attempted to create an account with the same UserID as another player. Additionally, a “games” table could be maintained to keep track of information about every game played, as shown below:

Date	Use rid	Difficulty	Win?

Here the “UserID” would serve as the foreign key, in order to ensure that no user could have a score recorded without being signed up. Although recorded every single game played would make the database extremely large, there are several benefits to this schema.

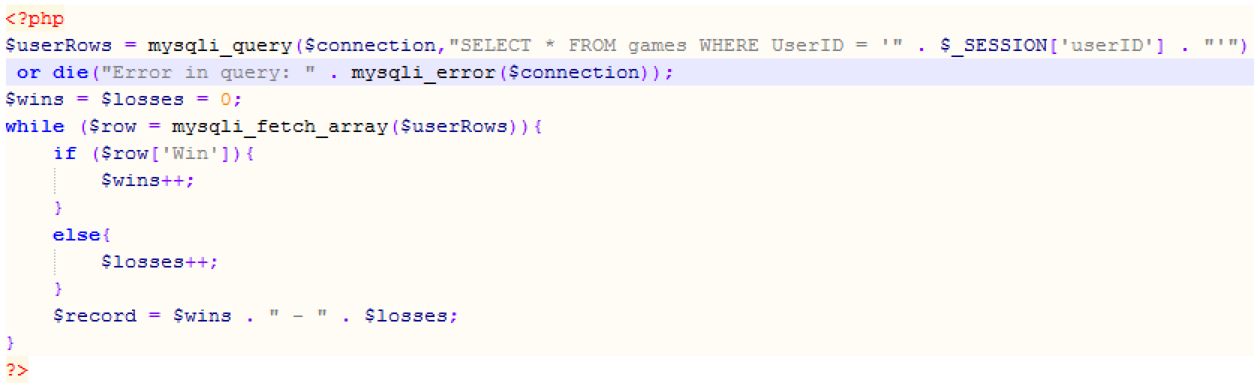
First, on the page where the game is played, the record of the player could be displayed using a query on the “games” table. Because the timestamp is recorded for each game, the player could choose to see his historic performance and compare it against other players. A leaderboard containing the userIDs of the players with the highest overall records could also be displayed on the game’s page. Finally, because the email addresses for each player are recorded in the “users” table, the server could send emails to players to alert them that they have a top record for the day, month, year, etc. Unfortunately, if a large number of games are played, the “games” table may become too large. In this case, some of the historic score capabilities would have to be sacrificed.

Shown below are example SQL queries which would create the discussed tables in a database:

"CREATE TABLE users (UserID VARCHAR(15) PRIMARY KEY, Password VARCHAR(15) NOT NULL, email VARCHAR(50),)"

"CREATE TABLE games (Timestamp TIMESTAMP NOT NULL, UserID VARCHAR(15) NOT NULL, Difficulty BYTE NOT NULL, Win YES/NO)"

The following PHP code could be used to find the overall record for a given player:



```php
<?php
$userRows = mysqli_query($connection,"SELECT * FROM games WHERE UserID = '" . $_SESSION['userID'] . "'")
 or die("Error in query: " . mysqli_error($connection));
$wins = $losses = 0;
while ($row = mysqli_fetch_array($userRows)){
    if ($row['Win']){
        $wins++;
    }
    else{
        $losses++;
    }
    $record = $wins . " - " . $losses;
}
?>
```

Blooms Taxonomy

Creation

To create our simple JavaScript Pong game we followed several steps.

1. HTML runner function which drives the JavaScript file
2. Create JavaScript file
3. Create animation canvas
4. Optimize JavaScript animations using requestAnimationFrame
5. Create and organize paddle, ball, player, and computer objects
6. Create helper functions such as render that multiple objects share
7. Implement prototyping to efficiently assign the helper functions to the objects
8. Remove global variables using the concept of closures
9. Implement hit detection logic for the paddles, top wall, and bottom wall
10. Design simple jQuery interface to show score and choose difficulties
11. Consider implementations of leaderboard database

Our creation process was iterative, much like a real world development cycle. Our first goal was to accomplish a working game, then move onto optimizations. This process helped our group become successful in our final implementation. Examples of our steps can be found in our attached source code.

Evaluation and Analysis

After implementing a rudimentary version of pong, evaluating where to implement optimizations was the majority of our evaluation and analysis. We decided to focus our optimizations on - code reuse, closures, and animations.

In our original implementation we had a lot of code that was duplicated. This was because each object had functions that were shared, but we decided to set the behavior of each object individually. Doing this helped us develop quickly, but needed to be refactored for maintainability. Our research concluded that using prototypes were the best solution. This allowed us to have one function being used by all of the objects, increasing our code reuse.

Additionally, we concluded that using global variables for some of our values was bad coding practice. We had experience with closures in lab, and decided it would be best to remove the global variables using the concept of closures. Similar to the code reuse we concluded that taking the extra time to strengthen our code would be a good way of increasing maintainability and scalability.

Finally, we were doing research on different animation techniques to find the best way to animate a canvas. We originally implemented using setTimeout just like was done in our lab. But our research showed that implementing it this way is not the most efficient. The decision to optimize our animation code was to create a better end product that uses the CPU more efficiently.

As stated many times, our choices in this portfolio were to create the best JavaScript code we were capable of. This meant keeping readability, maintainability, and scalability on our mind as we developed this application. We found this is the best way to learn, practice, and analyze a new topic in software design. We feel that our application follows these three principles, and it helped us produce a good product in the end.